Adaptive Garbage Collection for Modula-3 and Smalltalk

Richard Hudson[*] *Amer Diwan*[†]
Object Oriented Systems Laboratory
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

October 27, 1990

1 Introduction

Moon's ephemeral garbage collector [Moon, 1988] showed that gains can be made by segregating young objects from older objects. Ungar [Ungar, 1984] refined the criteria for segregation and showed how such segregation could benefit object oriented languages. Others later suggested that objects that last longer than one invocation of a program can be most efficiently dealt with using a persistent store.

Objects that aren't young and don't persist beyond one invocation of a program tend to behave in very application specific ways. Ungar notes that current generation-based systems perform poorly if objects live only into middle age but then die prior to program termination. We present scavenging mechanisms that adapt to the allocation patterns of the applications. Along the way we describe how to efficiently deal with multiple generations and remembered sets. We document the various policies that can be applied and parameters that can be manipulated in order to adjust how aggressively the mechanisms adapt. The algorithms described are of general interest to a variety of languages. This work was originally done to support Modula-3 and is currently being implemented for Smalltalk.

2 Overview

Garbage collectors that scavenge can consist of multiple generations. Each of these generations is divided into two semispaces. A scavenge consists of moving all reachable objects in the *from* semispace to the *to* semispace. After all reachable objects have been moved the semispaces change roles. When a generation is scavenged all generations younger than it are also scavenged. This insures that all reachable objects from younger to older generations will be encountered during a scavenge. Objects in generations that are reachable from older generations are recorded in a remembered set. Each generation has a remembered set associated with it. Remembered sets allow us to avoid scanning the older generations during a scavenge.

3 Memory Layout

Two critical pieces of the garbage collection mechanisms that need to be executed quickly are determining what generation an object is in and if that generation is younger than some other generation. These are critical because they need to be done for each store of a pointer. Ideally, we should be able to derive this information from the object addresses. To this end we have laid out generations in a right to left manner. That is to say younger generations always fall to the right of older generations. We have arranged it so that all generations are of the same size and that this size is a power of two. The generation of an address can

[*] University Computing Services; hudson@cs.umass.edu
[†] Department of Computer and Information Science; diwan@cs.umass.edu

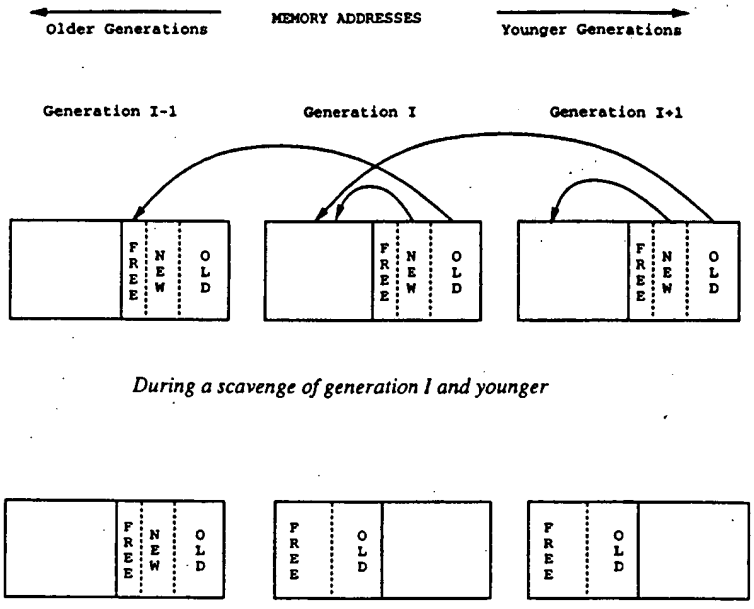

Figure 1: Promotion of objects

be determined by a bit shift operation. A simple comparison can then be used to determine if an address in an older generation refers to an object in a younger generation.

4 Design of the Adaptive Garbage Collector

Each object belongs to one of a variable number of *generations*. By convention, older generations have smaller generation numbers associated with them. The older generations are at lower addresses than the younger generations. The older a generation is the less often it is scavenged. One way to imagine this is to think about how an odometer works in a car. There is a 1's slot and it rotates 10 times for each rotation of the 10's slot. Scavenging of generations works in a similar manner. The younger a generation is, the more often it will be scavenged. To continue the odometer analogy the youngest generation scavenges items 10 times before the next older generation is scavenged.

Objects are promoted from one generation to the next based on the age of the object. The approximate age of an object can be determined by computing the generation it resides in. If we segregate those items that have survived one scavenge of their generation, then they can be promoted the next time that generation is scavenged. This technique is similar to one used by Wilson [Wilson and Moher, 1989].

Each time a generation is collected, every object in the *old* space is promoted. An object in $< i$, new> is copied into $< i$, old> during the scavenge. An object in $< i$, old> is promoted to generation $< i - 1$, new> if there is enough space in generation $i - 1$ to hold the new object. Note that $i - 1$ must have sufficient space in its currently active semispace to accept the promoted object. If there is no room in generation $i - 1$, then the promoted object is not promoted and a request to create a new generation between i and $i - 1$ is made. Figure 1 illustrates this.

When generation i can't promote an object, a new generation is created between generation i and generation $i - 1$. This happens when generation i next scavenges. Generations younger than i are copied in a manner that leaves the requested free space between generation i and $i - 1$. An object in $< i$, old> is

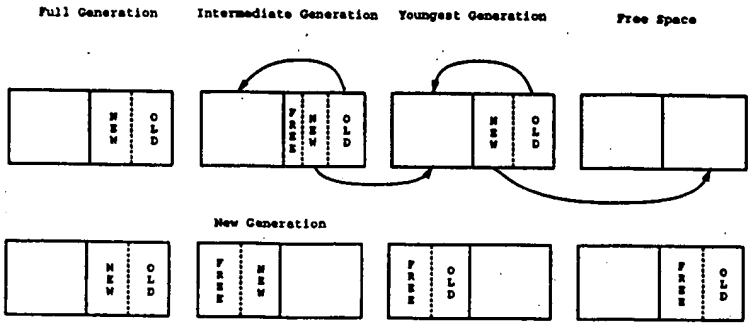

Figure 2: Migrating a new generation

promoted into i's other semispace. An object in $< i$, new> is copied into $i + 1$'s free semispace. An object in $< i + 1$, old> is also copied into $i + 1$'s free semispace. In other words for generations being scavenged objects that are to be promoted are copied into their own generation's *to* semispace and objects that aren't being promoted are copied into the *to* space of the next younger generation. Figure 2 demonstrates this. If more than one generation requests additional space this will be reflected in the semispace an object is copied into.

If scavenging a space frees up a lot of room in two adjacent generations, then these two generations can be merged. The free generation is migrated back, past the youngest generation. This is done analogously to inserting a generation. Having a variable number of fixed size generations that makes this approach adaptable, efficient and easy to reason about.

5 Store Buffer Processing

In this section we describe a scheme for maintaining remembered sets that minimizes the processing needed at the time of a store. The scheme removes duplicate remembered set entries. This minimizes the amount of memory used for bookkeeping and thus defers the need to scavenge. The scheme is adaptive in that it predicts the sizes of the remembered sets from previous cycles of this generation as well as from the size of the remembered set needed by objects likely to be promoted into this generation. In doing so it minimizes the processing between scavenges.

5.1 Sequential Store Buffers

A sequential store buffer is a fixed size buffer that is sequentially filled with the locations of addresses that might refer to objects in younger generations. Buffer overflow can be detected using a guard page. (A page is set to "no access" so that touching it reports an interrupt to the application). Adding an entry to the buffer during a store consists of incrementing a pointer and storing an address. If it can be arranged for the sequential store buffer pointer to reside in a register, this will be very fast.

When overflow of the sequential store buffer is detected, the values in the sequential store buffer are moved into the remembered set of the youngest generation. A tight iterative loop can be used to process the store buffer. During this move, duplicates in the sequential store buffer are removed. Other filters to detect and remove intragenerational pointers and non-pointers might be applied at this time. When a scavenge is done the items in a sequential store buffer are distributed into the appropriate remembered sets. One possible optimization is to try to make the sequential store buffer large enough so that it is emptied only during a scavenge. It is possible for items belonging in the remembered sets of older generations to be placed in the youngest generation during the emptying of the store buffer. These entries will be moved to

the proper remembered set during the next scavenge.

5.2 Remembered Set

If a pointer points into a younger generation, the location of that pointer is in the remembered set of the younger generation. Space for the remembered set is allocated at the beginning of *new* space. During a scavenge we create a new remembered set in *new* space. As the remembered set is created, values that belong in the remembered sets of older generations can be added to those remembered sets at this time.

The size to preallocate for a remembered set can be determined (for example) by examining the size of the remembered set of the generation about to promote objects into this generation.

6 Policy Decisions

When do we scavenge a generation? Some factors that could be taken into account are listed below:

- When a generation fills up, then scavenge.
- When a remembered set fills up, then scavenge.
- Scavenge more generations if the system is idle. White [White, 1980] suggested waiting until more memory can be acquired, others suggest waiting until a pause occurs during user input.
- Scavenge based on elapsed clock time (either CPU or real).
- Scavenge based on page fault frequency. This attempts to compact data and improve locality of reference.
- Decide when to scavenge based on the percentage of objects freed during the last several scavenges of this generation.
- Decide when to gc based on the percentage of objects freed during the last scavenge of the next youngest generation. If scavenging a neighbor was successful maybe scavenging its older neighbor would reclaim a lot of space.
- Decide to scavenge based upon the number of scavenges of the youngest generation.

We propose here that we utilize each of the above items in a function related to each generation. The parameters are fed to the function during each scavenge. Whenever the youngest generation decides it is time for a scavenge then each older generation is checked to see if it needs to be scavenged. This allows us a great deal of flexibility to investigate garbage collection policies.

7 Results

In order to better understand the performance implications of our store buffer design we have simulated the system and have run the Gabriel benchmark Destructive. Of all the Gabriel benches, Destructive most heavily exercises the storing of values. Without compile-time filtering we found that the additional overhead of our store buffer scheme, excluding emptying the sequential store buffer, was 10%. If the store buffer was much smaller and required emptying several times between scavenging, the overhead was 20%. Since Modula-3 is statically typed, the compiler can filter out the non-pointers at compile time. In Destructive this eliminates 94% of the stores and the time to process the store buffer becomes less than 1% of total processing time.

8 Conclusion

We have shown the mechanisms that are needed to manage scavenging garbage collectors with a number of generations. In the case of our store buffer processing we have given a result. Some of the new features we have investigated include:

- How to manage a variable number of fixed sized generations.
- Techniques that eliminate age and generation markers in the objects.
- Fast unconditional work at store time.
- Tight loop processing to perform filtering.

We have also discussed the "knobs" we think must be present in order to tune the system. We have suggested how such tuning might be accomplished dynamically.

References

[Moon, 1988] David Moon. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Austin, TX, Aug. 1988), ACM, pp. 235–246.

[Ungar, 1984] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, PA, Apr. 1984), ACM SIGPLAN Notices, ACM, pp. 157–167.

[White, 1980] Jon L. White. Address/memory management for a gigantic Lisp environment or, GC considered harmful. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Stanford, California, Aug. 1980), ACM, pp. 119–127.

[Wilson and Moher, 1989] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, LA, Oct. 1989), vol. 24(10) of *ACM SIGPLAN Notices*, ACM, pp. 23–35.